Using C++ for Low-Latency Systems

Patrice Roy
Patrice.Roy@USherbrooke.ca
Université de Sherbrooke

CppCon 2016

Who am I?

- Father of five (four girls, one boy), ages 21 to 3
- Feeds and cleans up after a varying number of animals
- Used to write military flight simulator code, among other things
- Full-time teacher since1998
- Works a lot with game programmers
- Incidentally, WG21 and WG23 member
- Involved in SG14

How we will proceed...

- Stick to standard, portable C++ as much as possible
- Examine features and practice
- Compare implementations
- Measure costs
  - Speed
  - Size
  - Programming effort
- Discuss together, trade ideas and tricks
- Big thank you to Matt Godbolt, whose wonderful https://godbolt.org was invaluable preparing this!

Latency?

- « Latency is the delay from input into a system to desired outcome » (WhatIs.com)
- « Latency is a time interval between the stimulation and response, or, from a more general point of view, a time delay between the cause and the effect of some physical change in the system being observed » (Wikipedia, from Webopedia)
- Latency tends to be associated with the response time following the occurrence of some event
  - Sometimes, we care about the moment when we start to react
  - Sometimes, we care about the moment when reaction completes

Latency?

- We seek to optimize according to the needs of a system or of its components
- Most applications seek good average response time
  - For example, the .NET and Java programming models tend in that direction
- Typically means accepting some costs, e.g.:
  - Garbage collection
  - JIT method compilation
  - Integrated profiling
  - etc.
- Even std::vector does this, in a sense
  - e.g.: `push_back()` is *amortized* constant time

Latency?

- Many latency-sensitive systems also care about worst-case performance
- Optimizing for the average case and optimizing for the worst-case can be antagonistic tasks
- Observer pattern over a network
  - Only get trafic when there's an actual message to transfer
  - Great best case (nothing happens!)
  - Great average case
  - Terrible worst-case (lots of events, network gets flooded)
- Polling much worse in general, but worst-case remains under the requesters' control

Latency?

- See **`00-Latency/pull.cpp`** for a polling-style example
- See **`00-Latency/push.cpp`** for an observer-style example
  - Both show an irregular producer
  - Polling-style consumes with regularity
    - The latency associated with cunsomption of a given message varies
  - Observer-style consumes with bursts
    - Latency is typically low when consuming a message
    - Computation requirements vary with the size of the bursts
    - Very good on best case and average case
    - Worst-case is unpredictable, and could cause latency-related problems
      - Can be alleviated by processing messages asynchronously

Low latency?

- Depends on the application domain
  - Reaction time to events in the low ms range
  - Professional audio output latency expected to be within 5ms
  - Sometimes in the sub-ms range
- Can involve a number of non-programming aspects
  - Co-location
  - Specialized or high-end hardware
  - Specialized (secret?) algorithms
- I hope you'll understand most of this is out of scope for us

Low latency?

- This class focuses on how to leverage C++ for low-latency applications
  - What costs you nothing
  - What costs you very little
  - What you should be careful with
  - How to use what the C++ standard offers to your advantage

Course Overview

- C++ for Low-Latency Systems
- Key Low/No Overhead C++ Features
- C++ Features to Use with Care in Low-Latency Systems
- Leveraging Standard Containers and Algorithms in Low-Latency Systems
- Concurrency and Parallel Programming
- Dynamic Memory Allocation
- Efficient Cache Usage
- Metaprogramming

Selected Principles of C++ Programming

- Language design
  - Zero-overhead abstraction (as much as possible)
  - There shall be no need for a lower-level language (except for a select few cases)
  - You don't pay for what you don't use (except for a select few cases)
- Programming practice
  - Do as the `int`s do (thank you Scott Meyers!)
  - All types get the same level of support
  - `const`-correctness

Selected Principles of C++ Programming

- Zero-overhead abstraction. Example using member functions and pointer arithmetic: https://godbolt.org/g/wHIHZq or **`01-Overview/manual-begin-end.cpp`**

```
#include <vector>
#include <algorithm>
#include <cstdio>
int main() {
  using namespace std;
  vector<int> v{ 2,3,5,7,11 };
  int arr[] { 2,3,5,7,11 };
  enum { N = sizeof(arr) / sizeof(arr[0]) };
  int sumA = 0;
  for(auto it = v.begin(); it != v.end(); ++it)
    sumA += *it;
  int sumB = 0;
  for(auto it = arr + 0; it != arr + N; ++it)
    sumB += *it;
  printf("%d %d", sumA, sumB);
}
```

Selected Principles of C++ Programming

- Zero-overhead abstraction. Example using free functions `std::begin()` and `std::end()`: https://godbolt.org/g/QK1e9j or **`01-Overview/std-begin-end.cpp`**

```
#include <vector>
#include <algorithm>
#include <cstdio>
int main() {
  using namespace std;
  vector<int> v{ 2,3,5,7,11 };
  int arr[] { 2,3,5,7,11 };
  int sumA = 0;
  for(auto it = begin(v); it != end(v); ++it)
    sumA += *it;
  int sumB = 0;
  for(auto it = begin(arr); it != end(arr); ++it)
    sumB += *it;
   printf("%d %d", sumA, sumB);
}
```

Selected Principles of C++ Programming

- Zero-overhead abstraction : example
  - Using pointer arithmetic, `begin()` and `end()` member functions, or `std::begin()` and `std::end()` free functions
    - https://godbolt.org/g/wHIHZq
    - https://godbolt.org/g/QK1e9j
- With the optimizer active, there is no cost to using such functions as `std::begin()` or `std::end()`
- They make code cleaner, more uniform… for free
  - And they require less typing!

Selected Principles of C++ Programming

- Zero-overhead abstraction : example

```
// classic array size trick
int arr[] { 2,3,5,7,11 };
enum { N = sizeof(arr) / sizeof(arr[0]) };
// C++11 is cool
template <class T, std::size_t N>
   constexpr
      std::size_t size(const T (&)[N]) {
         return N;
      }
```

- Both provide compile-time constants
- Calling `size(arr)` costs strictly nothing

Selected Principles of C++ Programming

- Do as the `int`s do : example

```
// works, but bad idea...
struct IMutex {
    virtual void lock() const = 0;
    virtual void unlock() const = 0;
    virtual ~IMutex() {}
};
IMutex *create(); // brittle, for many
                  // reasons...
```

Selected Principles of C++ Programming

- Do as the `int`s do : example

```
// works, a bit better...
#include <memory>
struct IMutex {
    virtual void lock() const = 0;
    virtual void unlock() const = 0;
    virtual ~IMutex() = default; // :)
};
// not brittle, but unpleasant...
std::unique_ptr<IMutex> create();
```

Selected Principles of C++ Programming

- Do as the `int`s do : example

```
// works, much better...
#include <memory>
class Mutex {
    struct Impl;
    std::unique_ptr<Impl> p;
public:
    Mutex();
    void lock() const;
    void unlock() const;
    ~Mutex();
};
```

Selected Principles of C++ Programming

- Do as the `int`s do : example

```
// now we're getting somewhere...
#include <memory>
class Mutex {
    struct Impl;
    std::unique_ptr<Impl> p;
public:
    Mutex();
    Mutex(Mutex&&);
    Mutex& operator=(Mutex&&);
    void lock() const;
    void unlock() const;
    ~Mutex();
};
```

  - Obviously, don't do this unless you have a *really* good reason : use `std::mutex`!

Exercises – Overview

- EX01-00: implement `Mutex` above to encapsulate a Win32 `HANDLE` or `CRITICAL_SECTION`, or a POSIX `pthread_mutex_t` or similar mechanism
  - What is the value of `sizeof(Mutex)` under your implementation?
- EX01-01: if you implement the `size()` free function above for other containers, in what cases can it be evaluated at compile-time?
- EX01-02: what are the costs to the *pImpl* approach used in the `Mutex` class above? Compared to what?

Measuring execution time

- There are many ways to measure execution time
  - Being very precise is tricky
  - Andrei Alexandrescu gave a very good talk on this topic toward the end of 2015, at Nokia
- For our purposes, it will typically suffice to do

$before \leftarrow now$

$f(args)$

$after \leftarrow now$

$elapsed \leftarrow after - before$

Measuring execution time

- We face technical challenges
  - We are fighting the optimizer
    - Our test code requires some side-effect to avoid being optimized away
  - We don't want to rewrite our test functions every time
    - We need something generic
  - We want client code to control the measurement's granularity
    - I know milliseconds are popular, but they're not always sufficient

Measuring execution time

- Our trick will be

```
#include <chrono>
#include <utility>
using namespace std::chrono;
template <class F, class ... Args>
   auto test(F f, Args&&... args) {
      auto pre = high_resolution_clock::now();
      auto res = f(std::forward<Args>(args)...);
      auto post = high_resolution_clock::now();
      return make_pair(res, post - pre);
   }
```

- Quite a few recent features in there…

Measuring execution time

- `<chrono>` utilities help represent and measure time
- There are three standard clocks (here, we use `high_resolution_clock`)
- Function `now()` returns a `time_point`
- Subtracting two instances of `time_point` yields a `duration`
- Types `time_point` and `duration` depend on the clock

```
#include <chrono>
#include <utility>
using namespace std::chrono;
template <class F, class ... Args>
   auto test(F f, Args&&... args) {
      auto pre = high_resolution_clock::now();
      auto res = f(std::forward<Args>(args)...);
      auto post = high_resolution_clock::now();
      return make_pair(res, post - pre);
   }
```

Measuring execution time

```
#include <chrono>
#include <utility>
using namespace std::chrono;
template <class F, class ... Args>
   auto test(F f, Args&&... args) {
      auto pre = high_resolution_clock::now();
      auto res = f(std::forward<Args>(args)...);
      auto post = high_resolution_clock::now();
      return make_pair(res, post - pre);
   }
```

- We'll take any function with any set of arguments as long as the combination compiles and makes sense
- We'll maintain such « qualities » a reference, ref-to-`const` when making the actual call

Measuring execution time

```
#include <chrono>
#include <utility>
using namespace std::chrono;
template <class F, class ... Args>
   auto test(F f, Args&&... args) {
      auto pre = high_resolution_clock::now();
      auto res = f(std::forward<Args>(args)...);
      auto post = high_resolution_clock::now();
      return make_pair(res, post - pre);
   }
```

- We want the code to work regardless of `f()`'s return type
- We want the code to work regardless of the clock's choice of `time_point`

Measuring execution time

```
#include <chrono>
#include <utility>
using namespace std::chrono;
template <class F, class ... Args>
   auto test(F f, Args&&... args) {
      auto pre = high_resolution_clock::now();
      auto res = f(std::forward<Args>(args)...);
      auto post = high_resolution_clock::now();
      return make_pair(res, post - pre);
   }
```

- We want client code to get both the result of the computation and the time elapsed

Measuring execution time

```
// ...
#include <thread>
#include <iostream>
using namespace std;
int main() {
  auto res = test([](int n) {
    this_thread::sleep_for(seconds{n});
    return 3;
  }, 2);
  cout << "Function ran for "
       << duration_cast<milliseconds>(
            res.second
          ).count()
       << " ms. and returned " << res.first
       << endl;
}
```

- See **`00-Latency/test.cpp`**

C++ for Low-Latency Systems

- We define what we mean by "low-latency"; discuss the similarities and differences with real-time systems; introduce worst-case execution time concepts and discuss measurement utilities and techniques
- Resilience-oriented aspects of C++ programming will be addressed here, but this will be a recurring theme throughout the course

static_assert

- When the compiler knows…
  - « Ouch, I just realized I wrote all this code as if `char` was signed… »
  - It does work when `char` is signed; we don't know what will happen if it's not…
- Or…
  - « I was sure `sizeof(int)` was supposed to be equal to `sizeof(int*)`… »

static_assert

- When the compiler knows…
  - « I wrote this cool generic function, but… I just realized it will behave strangely if we pass it something non-trivial… »

```
template <class T>
   unsigned char* raw_copy(const T &val, unsigned char *buf) {
      auto src = reinterpret_cast<const unsigned char*>(&val);
      copy(src, src + sizeof val, buf);
      return buf + sizeof val;
   }
```

- **Warning** : don't use such tricks blindly (we're very much into potential Undefined Behavior land)

static_assert

- `static_assert` is a beautiful thing
  - Zero cost
  - Reports errors without having to run the program
  - Documents expectations
  - Leads to more robust code
- `static_assert` tends to interact well with other nice features for low-latency systems
  - Compile-time constants
  - `constexpr` functions
  - traits
  - etc.

static_assert

```
static_assert(static_cast<char>(-1) < 0,
              "Signed char required");
// ...
static_assert(sizeof(int)==sizeof(int*),
              "Expect same size for pointers and ints");
// ...
#include <type_traits>
#include <algorithm>
template <class T>
   unsigned char* raw_copy(const T &src, unsigned char *buf) {
      static_assert(std::is_trivially_copyable_v<T>,
                    "Function requires bitwise copyable type");
      auto src = reinterpret_cast<const unsigned char*>(&val);
      std::copy(src, src + sizeof src, buf);
      return buf + sizeof val;
   }
```

- See **`02-NoCost/static_assert.cpp`**

static_assert

- Not all errors can be detected at compile-time, obviously
  - **`static_assert`** is great when applicable
- At run-time, a number of options exist
  - assertions through **`assert()`** or a similar mechanism
  - exceptions (yes, they're not *that* evil!)
  - specialized types such as **`optional<T>`**, **`expected<T,E>`**
    - See **`02-NoCost/maybe.cpp`**, **`02-NoCost/expected.cpp`**
    - Note :these are not « no cost » abstractions
  - returning error / success codes
  - returning **`std::pair`**, **`std::tuple`**
  - calling **`std::terminate()`** and friends
- It depends on context, really

variadic templates

- The more the compiler knows…
  - https://godbolt.org/g/HfCGaT or **02-NoCost/variadic_sum.cpp**
  - https://godbolt.org/g/Z6RvQu or **02-NoCost/manual_sum.cpp**
- There's no reason not to use such features
  - …if your compiler supports them
- They follow the zero-cost abstraction credo
- Another take on the same approach… It's difficult to generate simpler binaries than this :
  - https://godbolt.org/g/EfUSy1 or **02-NoCost/static_check_sum.cpp**

variadic templates

- Variadic multiple inheritance, manually
  - https://godbolt.org/g/3UV5PK
  - **`02-NoCost/multiple_inheritance_chunks.cpp`**

```
struct X { constexpr int f() const { return 3; } };
struct Y { constexpr int g() const { return 4; } };
struct Z { constexpr int h() const { return 5; } };
struct Chunk : X, Y, Z {
  constexpr Chunk() = default;
};
int main() {
  constexpr Chunk chunk{};
  static_assert(
    chunk.f() + chunk.g() + chunk.h() == 12,
    "Suspicious..."
  );
}
```

variadic templates

- Variadic multiple inheritance, « variadically »
  - https://godbolt.org/g/ad25xu
  - **02-NoCost/variadic_multiple_inheritance_chunks.cpp**

```
struct X { constexpr int f() const { return 3; } };
struct Y { constexpr int g() const { return 4; } };
struct Z { constexpr int h() const { return 5; } };
template <class ... P>
  struct Chunk : P... {
    constexpr Chunk() = default;
  };
int main() {
  constexpr Chunk<X,Y,Z> chunk{};
  static_assert(
    chunk.f() + chunk.g() + chunk.h() == 12,
    "Suspicious..."
  );
}
```

Automatic type deduction

- There are many who fear automatic type deduction. There are reasons for this
  - Suspicious usage

```
auto i = 0; // what's the point?
```

  - Losing track of what's going on

```
template <class T, class U>
   auto combine(T t, U u) { /* ... */ }
template <class T, class ... Ts>
   auto f(T arg, Ts ... ts) {
      return combine(arg, f(ts...));
   }
template <class T>
   auto f(T arg) { return arg; }
auto g() {
   return f("Wow", 3, Point{-1.0,2.5});
} // Ok, I'm lost
```

Automatic type deduction

- However, the C++ type system is pretty much static, and automatic type deduction is a compile-time mechanism
  - The following generate the same binary code

```
auto i = int{};
int i = 0;
```

  - So do the following

```
auto v = vector<string>{};
vector<string> v;
```

  - For such cases, automatic type deduction is a no-cost feature

Automatic type deduction

- Of course, using automatic type deduction, you have to know what you're doing

```
vector<string> v = { /* ... */ };
for(auto s : v) { // Oops, making copies of each element!
   // ...
}
for(auto &s : v) { // Each s taken as ref
   // ...
}
for(const auto &s : v) { // Each s taken as ref-to-const
   // ...
}
for(auto &&s : v) { // Each s taken as forwarding ref
   // ...
}
```

Automatic type deduction

- **auto** essentially lets the compiler deduce the type of a variable from the type of its initializer

```
// ...
vector<string> v;
vector<string>::iterator itA = begin(v);
auto itB = begin(v); // same thing
// ...
const vector<string> cv;
vector<string>::const_iterator citA =
    begin(v); // whew
auto citB = begin(v); // same thing
```

- In such a situation, auto simply saves typing

Automatic type deduction

- Note that `auto` can also simplify code maintenance

```
vector<string> f();
string g(string&&);
string h() {
  string res;
  auto v = f();
  for(auto &&s : v) res += g(std::move(s));
  return res;
}
```

- Here, `h()` depends on string, but `f()` could return `list<string>` or `deque<string>` without affecting the source code of `h()`

Automatic type deduction

- Sometimes, `auto` doesn't quite cut it
- Luckily, C++ has other no-cost automatic type deduction mechanisms

```
template <class R, class It, class T>
  R average(It first, It last, T acc = {}) {
    return static_cast<R>(
      accumulate(first, last, acc)
    ) / distance(first, last);
  }
```

- Correct call could be

```
int vals[] { 2,3,5,7,11 };
cout << average<float>(begin(vals),end(vals),0)
     << endl;
```

- The `0` is annoying in some cases. We could want a sensible default

Automatic type deduction

- If we thing the iterator's value_type is a sensible default, we can make things simpler to users, at no runtime cost

```
template <class R, class It,
          class T = remove_reference_t<
                       decltype(*declval<It>())
                    >>
  R average(It first, It last, T acc = {}) {
    return static_cast<R>(
      accumulate(first, last, acc)
    ) / distance(first, last);
  }
```

- Correct call could then be

```
int vals[] { 2,3,5,7,11 };
cout << average<float>(begin(vals),end(vals))
     << endl;
```

Automatic type deduction

- When, many new things at once
- Function **`declval<T>()`** is only usable in a compile-time context
  - It behaves as if it returned something of type `T`
- With `It` being `int*`, function `declval<It>()` would return `int*`
  - Thus, the type of `*declval<It>()` is `int&`
- Conpile-time operator **`decltype`** deduces the type of an expression
- We don’t want to accumulate on a reference
  - We use the `remove_reference_t` type trait to get from `int&` to `int`

Automatic type deduction

- Since C++14, automatic type deduction has gotten richer and more precise with **`decltype(auto)`**

```
template <class T>
   T&& pass_thru(T &&arg) { return arg; }
int main() {
  int n = 3;
  auto n0 = pass_thru(3); // n0 is int
  auto n1 = pass_thru(n); // n1 is int
  decltype(auto) n2 = pass_thru(3); // n2 is int
  decltype(auto) n3 = pass_thru(n); // n3 is int&
}
```

- `decltype(auto) x = expression;` stands for `decltype(expression) x = expression;`

Automatic type deduction

- All of this deduces types
  - It's all compile-time
  - It's all fair for low-latency code
- There is no technical reason not to use such features
  - There might be other reasons, of course
    - Company-local rules
    - Aesthetical preferences
- Well used, automatic type deduction can simplify coding at no runtime cost

Defaulted and deleted functions

- Deleted functions are useful to make code more secure

```
//
// careful : the « C++03-inspired » idiom
// translates awkwardly in the « C++11
// and later » world. See https://godbolt.org/g/hfX3fw
// for a short example
//
struct NonCopyable {
   NonCopyable(const NonCopyable&) = delete;
   NonCopyable& operator=(const NonCopyable&) = delete;
protected:
   NonCopyable() {
   }
   ~NonCopyable() {
   }
};
```

Defaulted and deleted functions

- Deleted functions are useful to make code more secure

```
//
// careful : the « C++03-inspired » idiom
// translates awkwardly in the « C++11
// and later » world. See https://godbolt.org/g/4oqBRG
// for a short example...
//
// See https://godbolt.org/g/Evg3aD for an even cooler one
//
struct NonCopyable {
   NonCopyable(const NonCopyable&) = delete;
   NonCopyable& operator=(const NonCopyable&) = delete;
protected:
   NonCopyable() = default;
   ~NonCopyable() = default;
};
```

Defaulted and deleted functions

- Note that the Boost-inspired `NonCopyable` class idiom interacts somewhat uncomfortably with move semantics
  - We cover move semantics below
  - See http://talesofcpp.fusionfenix.com/post-24/episode-eleven-to-kill-a-move-constructor by Agustín "K-ballo" Bergé for a detailed explanation of why this is so
    - It’s quite interesting, but not exactly our topic of interest today

Exercises – No-Cost Features

- EX02-00: write `avgdev(begin,end)` to compute generically the average deviation of the half-open sequence *[begin,end)*
  - Use the following form: $\sqrt{\frac{(x-\bar{x})^2}{(n-1)}}$
  - Does your solution work well with a sequence of `short`?
  - Does your solution work well with a sequence of `int`?
  - Does your solution work well with a sequence of `float`?
  - How do you handle the case where `distance(begin,end)==1`?

Exercises – No-Cost Features

- EX02-01: write a compile-time `average()` free function that takes a variadic number of arguments
  - How do you handle the no argument case?
- EX02-02: (trickier) can you write a class that has a variadic number of parents, each of which exposes a member function named `f()` but that takes different arguments, in such a way that all `f()` functions are exposed through an instance of the child class?

Key Low/No Overhead C++ Features

- Some C++ features have strong upsides and little to no downsides for low-latency system development
- In this section of the course, we will explore and use some of the most relevant ones, including
  - `unique_ptr`
  - `make_unique`
  - `constexpr`
  - Type traits
  - `enable_if`
  - Move semantics
  - Perfect forwarding

unique_ptr

- `unique_ptr<T>` is a beautiful thing
  - No size overhead compared to `T*` for the common case
    - No size overhead compared to `T*` for some uncommon cases too
  - No speed overhead compared to `T*` for most cases
    - For an example, see http://ideone.com/hYVZ4K or **`03-LowCost/unique_ptr_comparisons.cpp`**
    - Pay attention to the difference between the naïve and the realistic usage of `unique_ptr` (it might or might not make a difference depending on your compiler)
  - Simplifies coding
  - Simplifies memory management

unique_ptr

```
#include <memory>
using namespace std;
static_assert(sizeof(double*)==sizeof(unique_ptr<double>),
              "...");
class X {
  ~X() = default;
  friend struct Y;
};
struct Y {
   void operator()(const X*p) { delete p; }
};
static_assert(sizeof(X*)==sizeof(unique_ptr<X,Y>), "...");
int main() {
    // X x; // no, X::~X() is private
    unique_ptr<X,Y> p{ new X }; // fine
}
```

- See **03-LowCost/unique_ptr_functor_deleter.cpp**

unique_ptr

```
#include <cstdio>
class X {
  int val;
public:
  X(int val) : val{val} { }
  int f(int n) const { return n + val; }
};
auto f(X *p) {
  int arr[] { 2,3,5,7,11 };
  int sum = 0;
  for(auto n : arr) sum += p->f(n);
  delete p;
  return sum;
}
int main() {
  X *p{ new X{ 3 }};
  std::printf("%d", f(p));
}
```

- See https://godbolt.org/g/KY1rc0 or **03-LowCost/ptr_manual_mgmt.cpp**

unique_ptr

```
#include <cstdio>
class X {
  int val;
public:
  X(int val) : val{val} { }
  int f(int n) const { return n + val; }
};
auto f(X *p) {
  int arr[] { 2,3,5,7,11 };
  int sum = 0;
  for(auto n : arr) sum += p->f(n);
  delete p;
  return sum;
}
int main() {
  std::printf("%d", f(new X{ 3 }));
}
```

- See https://godbolt.org/g/unj76T or **`03-LowCost/ptr_manual_mgmt_2.cpp`**

unique_ptr

```
#include <memory>
#include <cstdio>
class X {
  int val;
public:
  X(int val) : val{val} { }
  int f(int n) const { return n + val; }
};
auto f(std::unique_ptr<X> p) {
  int arr[] { 2,3,5,7,11 };
  int sum = 0;
  for(auto n : arr) sum += p->f(n);
  return sum;
}
int main() {
  std::unique_ptr<X> p{ new X{ 3 }};
  std::printf("%d", f(std::move(p)));
}
```

- See https://godbolt.org/g/0K4ENr or **`03-LowCost/ptr_unique_ptr_mgmt.cpp`**

unique_ptr

```
#include <memory>
#include <cstdio>
class X {
  int val;
public:
  X(int val) : val{val} { }
  int f(int n) const { return n + val; }
};
auto f(std::unique_ptr<X> p) {
  int arr[] { 2,3,5,7,11 };
  int sum = 0;
  for(auto n : arr) sum += p->f(n);
  return sum;
}
int main() {
  std::printf(f(std::unique_ptr<X>{ new X{ 3 }}));
}
```

- See https://godbolt.org/g/k1y69r or **`03-LowCost/ptr_unique_ptr_mgmt_2.cpp`**

unique_ptr – Costs (speed)

- There is a cost to moving from this…

```
void f(vector<int*> v) {
  vector<int*> w;
  w.reserve(v.size());
  for(auto & p : v) {
    w.push_back(p); // two MOV + push_back code
    p = {};         // if we count this one
  }
}
```

- … to this:

```
void f(vector<unique_ptr<int>> v) {
  vector<unique_ptr<int>> w;
  w.reserve(v.size());
  for(auto & p : v)
    w.push_back(std::move(p)); // three MOV + push_back...
}
```

See http://ideone.com/dZaDcy or **03-LowCost/vector_unique_ptr_cost.cpp**

unique_ptr – Costs (size)

- In some cases, `unique_ptr` has to store its deleter
  - Stateful deleters
    - Example: an object that could use various finalization strategies depending on context, and stores them internally
  - Function pointers
    - The type is not enough to distinguish them
- Then, `sizeof(unique_ptr<T>)>sizeof(T*)`
  - Probably by the size of a function pointer, in practice, but it's a non-zero cost in terms of space

unique_ptr – Costs (size)

```
class X {
  ~X(); // private
public:
   void destroy() const { delete this; }
  // ...
};
void destroyer(const X *p) { if (p) p->destroy(); }
// ...
int main() {
  // decltype(&destroyer) is void(*)(const X*)
  unique_ptr<X,decltype(&destroyer)> p {
    new X, destroyer
  };
  static_assert(sizeof p > sizeof(X*), "...");
  // ...
}
```

- See **03-LowCost/unique_ptr_comparisons_2.cpp**

make_unique

- It's typically a bad idea to give more than one responsibility to a single class
  - Code gets messy…
- Example class with a single responsibility
  - http://ideone.com/84xEzH or **`03-LowCost/example_single_responsibility.cpp`**
  - 47 lines of code
  - Relatively straightforward
- Example class with two responsibilities
  - http://ideone.com/5is0as or **`03-LowCost/example_two_responsibilities.cpp`**
  - 77 lines of code
  - Need for exception handling

make_unique

- It's a bit nicer with `unique_ptr`
- Example class with a single responsibility (revisited)
  - http://ideone.com/YMhIl3 or **`03-LowCost/example_single_responsibility_revisited.cpp`**
  - 45 lines of code (could be smaller)
  - Still relatively straightforward
- Example class with two responsibilities (revisited)
  - http://ideone.com/JPxnRA or **`03-LowCost/example_two_responsibilities_revisited.cpp`**
  - 58 lines of code
  - Need for explicit exception handling disappeared
  - Still, pay attention to some of the constructors

make_unique

- Take this one

```
TwoResponsibilities
  (const string &s0, const string &s1)
     : p { new string { s0 } } {
  q = unique_ptr<string>{ new string { s1 } };
}
```

- If we wrote it this way instead…

```
TwoResponsibilities
  (const string &s0, const string &s1)
    : p { new string { s0 } }, q { new string { s1 } }
{
}
```

- … there would be a leak risk if both calls to `new` occur before both `unique_ptr` constructors

make_unique

- This is where `make_unique()` shines

```
TwoResponsibilities
   (const string &s0, const string &s1)
     : p { make_unique<string>(s0) },
       q { make_unique<string>(s1) }
{
}
```

- Each `make_unique()` call either completes or does not
- If a `make_unique()` call completes, it yields a fully constructed `unique_ptr`
  - No leak!

make_unique

- What does `make_unique()` cost?
  - Small program relying on `make_unique()`
    - https://godbolt.org/g/fluC6r or **`03-LowCost/small_program_make_unique.cpp`**
  - Small program relying on `unique_ptr` without `make_unique()`
    - https://godbolt.org/g/I2RKAN or **`03-LowCost/small_program_unique_ptr.cpp`**
  - Small program that does manual memory management
    - https://godbolt.org/g/BCPKab or **`03-LowCost/small_program_manual_ptr.cpp`**
- Visibly, `make_unique()` only brings sanity to your code

constexpr

- One of the most powerful C++11 features
- Made more powerful through C++14 refinements
- Transforms some traditionally runtime computations into compile-time computations, recognizable as such

```
constexpr unsigned long long facto(int n) {
   return n == 0 || n == 1 ?
          1ULL : n * facto(n-1); // yes :)
}
int main() {
   // array of 120 floats
   float arr[facto(5)]{ };
}
```

constexpr

- `constexpr` goes well with user-defined literals (UDL)
  - UDLs don't have to be constexpr, but they can be
- A slightly ridiculous illustration follows

```
constexpr unsigned long long facto(int n) {
   return n == 0 || n == 1 ?
          1ULL : n * facto(n-1);
}
constexpr auto operator"" _fac
   (unsigned long long n) {
   return facto(static_cast<int>(n));
}
int main() {
   // array of 120 floats
   float arr[5_fac]{ };
}
```

constexpr

- The `constexpr` keyword is pure optimizer candy…

```
#include <type_traits>
class exact{}; class floating_point{};
template <class T>
   constexpr T absolute(T val) {
      return val < 0? -val : val;
   }
template <class T>
   constexpr T threshold = static_cast<T>(0.000001);
template <class T>
  constexpr bool close_enough(T a, T b, exact){ return a==b; }
template <class T>
  constexpr bool close_enough(T a, T b, floating_point) {
    return absolute(a - b) <= threshold<T>;
  }
```

constexpr

- See http://ideone.com/eK3GW6 or **03-LowCost/close_enough.cpp**

```
template <class T>
   constexpr bool close_enough(T a, T b) {
      return close_enough(a, b, std::conditional_t<
        std::is_floating_point<T>::value,
        floating_point, exact
      >{});
   }
int main() {
   static_assert(close_enough(3,3), "...");
   static_assert(!close_enough(3.1,3.0), "...");
   static_assert(close_enough(3.000000000001,3.0),
                 "...");
}
```

- To see the generated assembly : https://godbolt.org/g/zlUW5V

Parenthesis – Tag dipatching

- Write functions with same name but different signatures
- Ensure signatures differ on a single type (the tag, typically an empty class)
- Find a compile-time way to instantiate the correct tag type in order to call the appropriate function
  - Overload resolution is performed at compile time!

constexpr

- Another example (1/3) :

```
class invalid_grade {};
class Grade {
public:
  using value_type = int;
private:
  value_type val;
  static constexpr value_type minval() { return 0; }
  static constexpr value_type maxval() { return 100; }
  static constexpr bool is_valid(value_type candidate) {
    return minval() <= candidate && candidate <= maxval();
  }
  static constexpr value_type validate(value_type candidate) {
    return is_valid(candidate)? candidate
                              : throw invalid_grade{};
  }
  // ...
```

constexpr

- Another example (2/3):

```
  // ...
public:
  constexpr value_type value() const noexcept { return val; }
  static constexpr value_type passing_grade() { return 60; }
  constexpr Grade() noexcept : val { minval() } {
  }
  constexpr Grade(value_type val) : val { validate(val) } {
  }
  constexpr bool operator==(const Grade &g) const noexcept {
    return close_enough(value(), g.value());
  }
  constexpr bool operator!=(const Grade &g) const noexcept {
    return !(*this == g);
  }
};
```

constexpr

- Another example (3/3) :

```
int main() {
  static constexpr int
    PASSING_GRADE = 60;
  static_assert(
    close_enough(PASSING_GRADE,
                 Grade::passing_grade()),
    "...");
}
```

- See http://ideone.com/QTL4Su or **`03-LowCost/grade.cpp`**
- For the generated binaries : https://godbolt.org/g/uQPS7f
- Class `Grade` is ROM-able!

constexpr

- Compile-time lookup tables. A thing of beauty (thanks to Peter Somerlad for inspiration) for C++14
  - https://godbolt.org/g/oscD6V (no optimization)
  - https://godbolt.org/g/8fgi2R (with -O2)
  - Sources: **`03-LowCost/lookup_table.cpp`**
- `constexpr` is a whole world to explore, for those who are into no-cost computation

Type traits

- Type traits are in-code documentation on types
- They're compile-time, and can be used as such
- An example of optimization through traits can be found at http://ideone.com/PyOco4
  - See also **`03-LowCost/lexical_cast.cpp`**
  - The code uses traits to guide the compiler through the appropriate functions
  - Other examples have appeared before (see function `close_enough()` for example)

Type traits

- Old school Andrei Alexandrescu-inspired `is_convertible<S,D>` (or: how to leverage the compiler)

```
template <class S, class D>
   class is_convertible {
      using yes= char;
      struct no { char _ [3]; };
      static yes test(D);
      static no test(...);
      static S gen();
   public:
      enum {
         value = sizeof(test(gen())) == sizeof(yes)
      };
   }; // use std::is_convertible, not this, please!
```

Type traits

- Slighty less old school `is_convertible<S,D>`

```
template <class S, class D>
   class is_convertible {
      using yes= char;
      struct no {
         char _ [3];
      };
      static yes test(D);
      static no test(...);
   public:
      enum {
         value = sizeof(test(declval<S>())) ==
                 sizeof(yes)
      };
   }; // use std::is_convertible, no this, please!
```

Type traits and constexpr

- We've seen `close_enough()` with tag dispatching. Alernative (more modern) version:

```
#include <type_traits>
template <class T> constexpr T absolute(T val) {
   return val < 0? -val : val;
}
template <class T>
   constexpr T threshold = static_cast<T>(0.000001);
template <class T>
   std::enable_if_t<std::is_floating_point_v<T>, T>
      constexpr close_enough(T a, T b) {
         return absolute(a - b) <= threshold<T>;
      }
template <class T>
   std::enable_if_t<!std::is_floating_point_v<T>, T>
      constexpr close_enough(T a, T b) {
         return a==b;
      }
```

- See **`03-LowCost/close_enough_enable_if.cpp`**

enable_if

- `enable_if<V,T>` can be used to exclude some functions from the overload set for a given call
  - Overload resolution is performed at compile time!
- Has some aesthetical (and some technical!) advantages over tag dispatching
- Tag dispatching relies on a « front-end » function calling other « back-end » functions
  - `enable_if` simply excludes some functions from the overload set
    - Relies on SFINAE
  - Ideally, all candidate functions for a given call are removed, except for one (the correct one)

Movement, RVO, NRVO

- Take this function:

```
vector<double> f(vector<double> v) {
  transform(begin(v), end(v),
            begin(v), [](double x) {
    return sqrt(x);
  });
  return v;
}
```

- From a C++03 perspective, this looks like heresy…

movement, perfect forwarding, RVO, NRVO

- Executing this naively, we get
  - http://ideone.com/NqlNwU
  - **`03-LowCost/f_of_v_copy.cpp`**
  - By naively, I mean `v=f(v)` where `v` is a large `vector<double>`
- If we use move semantics, recognizing the fact that `f()` does not need to copy `v`, we get
  - http://ideone.com/61Hqtf
  - **`03-LowCost/f_of_v_move.cpp`**
  - In this case, we use `v=f(std::move(v))`
  - Since the contents of `v` are going to be replaced after the call to `v`, there's not need to keep them intact

Movement, RVO, NRVO

- What if we change `f()` to the following, writing « Old School C++ »?

```
void f(vector<double> &v) {
   transform(begin(v), end(v),
             begin(v), [](double x) {
     return sqrt(x);
   });
}
```

- Well, we get
  - http://ideone.com/nRDP1O
  - **`03-LowCost/f_of_v_ref.cpp`**
- Does that surprise you?

Movement, RVO, NRVO

- The « clean code » is `v = f(v);`
    - It used to be seen as inefficient, due to the copy of argument `v`
- Pass-by-ref tends to complicate optimizers' jobs
    - It's easier to reason on an object no-one else than you can access
- With move semantics, we get the best of both worlds
    - `v=f(std::move(v));`
    - There's always a single « owner » of `v`
- Clean code becomes fast code

Movement, RVO, NRVO

- Here's class `Noisy`, a good friend
  - Thanks to STL for the name and the technique
  - See **`03-LowCost/noisy.h`**

```
struct Noisy {
   Noisy() { cout << "Noisy()" << endl; }
   Noisy(const Noisy&) {
      cout << "Noisy(const Noisy&)" << endl;
   }
   Noisy(Noisy&&) { cout << "Noisy(Noisy&&)" << endl; }
   Noisy& operator=(const Noisy&) {
      cout << "operator=(const Noisy&)" << endl; return *this;
   }
   Noisy& operator=(Noisy&&) {
      cout << "operator=(Noisy&&)" << endl; return *this;
   }
   ~Noisy() { cout << "~Noisy()" << endl; }
};
```

Movement, RVO, NRVO

```
//
// ... Noisy ...
//
template <class T>
   T f(T arg) {
      return arg;
   }
int main() {
   Noisy n;
   n = f(n);
}
```

Movement, RVO, NRVO

```
//
// ... Noisy ...
//
template <class T>
   T f(T arg) {
      return arg;
   }
int main() {
   Noisy n;
   n = f(n);
}
```

- `Noisy()`
- `Noisy(const Noisy&)`
- `Noisy(Noisy&&)`
- `~Noisy()`
- `operator=(Noisy&&)`
- `~Noisy()`
- `~Noisy()`

Movement, RVO, NRVO

```
//
// ... Noisy ...
//
template <class T>
   T f() {
      return{};
   }
int main() {
   Noisy n;
   n = f<Noisy>();
}
```

Movement, RVO, NRVO

```
//
// ... Noisy ...
//
template <class T>
   T f() {
      return{};
   }
int main() {
   Noisy n;
   n = f<Noisy>();
}
```

- `Noisy()`
- `Noisy()`
- `operator=(Noisy&&)`
- `~Noisy()`
- `~Noisy()`

Movement, RVO, NRVO

```
//
// ... Noisy ...
//
template <class T>
   T f(T arg) {
      return arg;
   }
int main() {
   Noisy n =
      f(Noisy{});
}
```

Movement, RVO, NRVO

```
//
// ... Noisy ...
//
template <class T>
   T f(T arg) {
      return arg;
   }
int main() {
   Noisy n =
      f(Noisy{});
}
```

- `Noisy()`
- `Noisy(Noisy&&)`
- `~Noisy()`
- `~Noisy()`

Movement, RVO, NRVO

```
//
// ... Noisy ...
//
template <class T>
   T f() {
      return{};
   }
int main() {
   Noisy n =
      f<Noisy>();
}
```

Movement, RVO, NRVO

```
//
// ... Noisy ...
//
template <class T>
   T f() {
      return{};
   }
int main() {
   Noisy n =
      f<Noisy>();
}
```

- Noisy()
- ~Noisy()

Movement, RVO, NRVO

- Through Return-Value Optimization (RVO), the compiler can remove entire temporaries from the program
- Used to be optional; a slight problem
  - If this optimization is optional, then it's noticeable (not « as if »)
- RVO becomes mandatory with C++17

Perfect forwarding

- Let's suppose we want this program to work

```
int main() {
   print("I love my teacher",
         3, 3.14159);
}
```

Perfect forwarding

- A naïve, working solution might be **`03-LowCost/variadic_print_naive.cpp`**

```
#include <iostream>
using namespace std;
template <class T>
void print(const T &arg) {
   cout << arg << ' ';
}
template <class T, class ... Ts>
void print(const T &arg, Ts ... args) {
   print(arg);
   print(args...);
}
int main() {
   print("I love my teacher", 3, 3.14159);
}
```

Perfect forwarding

- A variation might be **03-LowCost/variadic_printsz_naive.cpp**

```
#include <iostream>
using namespace std;
template <class T>
void printsz(const T &arg) {
   cout << sizeof arg << ' ';
}
template <class T, class ... Ts>
void printsz(const T &arg, Ts ... args) {
   printsz(arg);
   printsz(args...);
}
int main() {
   printsz("I love my teacher", 3, 3.14159);
}
```

- Prints something like `18 4 8`

Perfect forwarding

- If we change the order of arguments, we might get **`03-LowCost/variadic_printsz_oops.cpp`**

```
#include <iostream>
using namespace std;
template <class T>
void printsz(const T &arg) {
   cout << sizeof arg << ' ';
}
template <class T, class ... Ts>
void printsz(const T &arg, Ts ... args) {
   printsz(arg);
   printsz(args...);
}
int main() {
   printsz(3, "I love my teacher", 3.14159);
}
```

- Prints something like **`4 4 8`**… What's going on?

Perfect forwarding

- A variation might be **`03-LowCost/variadic_printsz_ok.cpp`**

```
#include <iostream>
using namespace std;
template <class T>
void printsz(T &&arg) {
   cout << sizeof arg << ' ';
}
template <class T, class ... Ts>
void printsz(T &&arg, Ts &&... args) {
   printsz(std::forward<T>(arg));
   printsz(std::forward<Ts>(args)...);
}
int main() {
   printsz(3, "I love my teacher", 3.14159);
}
```

- Prints something like **`4 18 8`**. Much better!

Perfect forwarding

- Relaying arguments with the appropriate type can be a matter of correctness
- It can also be a matter of optimization
  - Keep moveable things moveable
  - Avoid some unnecessary copies
- Forwarding references carry information with them
- A named argument that used to be moveable loses that « moveability »
  - This important characteristic can be regained through proper forwarding

Exercises – Low-Cost Features

- EX03-00: write function `is_impure()` that takes a non-void function `f` and a variadic pack of arguments, and returns `true` only if calling `f()` with these arguments twice returns different values. Note that this function is sufficient but not necessary, as it can yield false negatives
  - Should you apply forwarding to the arguments?
  - Suppose you want to store the return values of calls to `f` in local variables to your `is_impure()` functions before comparing them. How many ways can you write the type of these variables?

Exercises – Low-Cost Features

- EX03-01: write function `how_many(dt,f,args)` which computes how many calls to `f` with variadic argument pack `args` can be performed without exceeding delay `dt`
  - Express `dt` in terms of `std::chrono` measurement units, e.g.: `2s`, `milliseconds{200}`, `5'000'000us`, etc.
  - Is forwarding of arguments a good idea in this case?
  - How do you handle the case where `f(args)` takes more than `dt` to complete?

Exercises – Low-Cost Features

- EX03-02: write the necessary code for the following program to compile and execute appropriately

```
int main() {
   constexpr Temperature<Celsius> cels = 0;
   // conversion from °C to °F
   constexpr Temperature<Fahrenheit> fahr=cels;
   static_assert(
      close_enough(fahr.value(),32.0),
      "Hmm"
   );
   static_assert(
      Temperature<Kelvin>{0} < cels,
      "Hmm"
   );
   cout << fahr << endl; // 32 F, at runtime
}
```

Exercises – Low-Cost Features

- EX03-03: write the necessary code for the following program to compile and execute appropriately

```
int main() {
   constexpr auto cels = 0_Cels;
   // conversion from °C to °F
   constexpr Temperature<Fahrenheit> fahr=cels;
   static_assert(
      close_enough(fahr.value(),32.0),
      "Hmm"
   );
   static_assert(0_Kelv < cels, "Hmm");
   cout << fahr << endl; // 32 F, at runtime
}
```

C++ Features to Use with Care in Low-Latency Systems

- There are C++ features that can be used for low-latency systems but with particular care only, as they have costs that could be higher than their benefits
- We will take a reasoned, measurement-based approach to build an informed judgement
- C++ features used will include
  - Run-time polymorphism and indirect function calls
  - `shared_ptr`
  - `make_shared`
  - Exceptions (yes, really)
  - Multiple inheritance
  - RTTI and used-defined replacements thereof
- We will also discuss the role of `noexcept` in such systems

Run-time polymorphism and indirect function calls

- Indirect function calls are fast…
  - … in many low-latency systems, they are called extremely often without harmful effects
- …but they have non-zero cost
- Let’s compare two types of callback systems

Run-time polymorphism and indirect function calls

- Callback through a functor or a λ

```
template <class It, class Pred>
  auto callback_ftor
    (It b, It e, Pred pred) {
      return count_if(b, e, pred);
    }
```

Run-time polymorphism and indirect function calls

- Callback through a function pointer

```
template <class It, class R, class A>
  auto callback_fctn
    (It b, It e, R(*pred)(A)) {
      return count_if(b, e, pred);
    }
```

Run-time polymorphism and indirect function calls

- Some test code
- See **`04-WithCare/callbacks.cpp`**

```
bool is_even(int n) { return n % 2 == 0; }
int main() {
   enum { N = 10'000'000 };
   vector<int> v(N);
   iota(begin(v), end(v), 1);
   auto r0 = test([&]() {
      return callback_fctn(begin(v), end(v), is_even);
   });
   // ...
   auto r1 = test([&]() {
      return callback_ftor(begin(v), end(v), [](int n) {
         return n % 2 == 0;
      });
   });
   // ...
}
```

shared_ptr

make_shared

exceptions

noexcept

Multiple inheritance

RTTI

- …and used-defined replacements thereof

Leveraging Standard Containers and Algorithms in Low-Latency Systems

- Some companies rewrite some or all of the standards containers due to a belief that their needs cannot be satisfied by the tools this library offers. We will explore ways to get the most out of the containers and the algorithms provided by the standard library, and examine ways to enhance them when appropriate

Concurrency and Parallel Programming

- Low-latency systems concerns overlap those of parallel and concurrent systems in different ways: reducing reliance on blocking operations and synchronization facilities, for example, is a strong trend in these two areas.
- C++ features used will include C++ 11/14 threading and atomics tools, but with an emphasis on usage patterns that allow components to react in a timely manner to events

thread

- with an emphasis on usage patterns that allow components to react in a timely manner to events

future

- with an emphasis on usage patterns that allow components to react in a timely manner to events

atomics

- with an emphasis on usage patterns that allow components to react in a timely manner to events

Dynamic Memory Allocation

- C++ lets programmers manage memory allocation manually in various and interesting ways. We will explore ways to use these mechanisms to our advantage when building low-latency systems.
- C++ features used will include specializing operators new and delete in many ways, understanding placement new, and how to write allocators before and since C++ 11.

overloading new

Placement new

Object lifetime

- Construction
- Placement new

arena

Efficient Cache Usage

- There have been a number of interesting presentations on cache memory usage and its impact on program performance. We will explore the nuances of this question with examples.
- C++ features used will include STL containers and algorithms, alignment control mechanisms, as well as parallel and concurrent programming utilities

Metaprogramming

- Of course, when it is practical to do so, the fastest programs are those that do nothing. We will explore some ways to combine metaprogramming techniques with small programs in order to move computation to compile time when there are benefits in so doing